International Journal of Electronics and Communication Engineering (IJECE)
ISSN(P): 2278-9901; ISSN(E): 2278-991X
Vol. 3, Issue 2, Mar 2014, 33-48

International Academy of Science, Engineering and Technology
Connecting Researchers; Nurturing Innovations

A DETAILED ANALYSIS OF LAYERED DEPTH IMAGES

G. PINAKAPANI[1] & T. VENKATESWARLU[2]

[1]Department of Electrical and Electronics Engineering, SV University, Tirupati, Andhra Pradesh, India

[2]Department of Electronics and Communication Engineering, SV University, Tirupati, Andhra Pradesh, India

ABSTRACT

Here we implemented a novel rendering method based on images converted from frames. Compared to other techniques, first method warps sprites with smooth surfaces which represent depth without gaps. Another method performs warping for more general scenes depending upon the halfway representation named LDI.LDI sight depends on single input camera view, but depends on multiple pixels in line of sight. Depends on depth complexity, size of portrayal changes. McMillan's warp ordering algorithm can be implemented because of single image coordinate system of LDI, resulting back to front order of pixels drawn in output image. Alpha compositing can be done effectively without depth sorting and no usage of z-buffer, so splitting becomes best solution for re-sampling problem.

KEYWORDS: Plane Filtered, 3D Mapping, Depth Images, FSPF

1. INTRODUCTION

Applications like 3D mapping and reconstruction, shape analysis, pose tracking and object recognition can potentially benefit from this sensor modality. However, given that indoor mobile robots have limited onboard computational power it is infeasible to process the complete 3D point clouds in real time and at full frame rates (e.g. the Microsoft Kinect sensor produces 9.2M3D pts/sec). Feature extraction, and in particular, geometric feature extraction is therefore the natural choice for abstracting the sensor data. However, noisy sensing and the presence he sprite approximation's fidelity to the correct new view is highly dependent on the geometry being represented. In particular shown as orange points, corresponding convex polygons shown in blue.

The complete 3D point cloud is overlaid as translucent grey for reference. (b) Scene polygon set generated by merging polygons from 15 consecutive depth image frames of geometric outliers (objects amidst the geometric features that do not match the geometric model of the features) provide additional challenges to the task of geometric feature extraction. We introduce the Fast Sampling Plane Filtering (FSPF) algorithm that samples the depth image to produce a set of “plane filtered” points corresponding to planes, the corresponding plane parameters (normals and offsets) and the convex polygons in 3D to fit these plane filtered points.

The FSPF algorithm meets the following goals:

- Reduce the volume of the 3D point cloud by generating a smaller set of “plane filtered” 3D points
- Compute convex polygons to fit the plane filtered points
- Iteratively merge convex plane polygons without maintaining a history of all observed plane filtered points
- Perform all of the above in real time and at full frame rates

This paper introduces two new extensions to overcome both of these limitations. The first extension is primarily applicable to smoothly varying surfaces, while the second is useful primarily for very complex geometries. Each method provides efficient image based rendering capable of producing multiple frames per second on a PC. In the case of sprites representing smoothly varying surfaces, we introduce an algorithm for rendering *Sprites with Depth*. The algorithm first forward maps (i.e., warps) the depth values themselves and then uses this information to add parallax corrections to a standard sprite renderer. For more complex geometries, we introduce the *Layered Depth Image*, or LDI, that contains potentially multiple depth pixels at each discrete location in the image. Instead of a 2D array of depth pixels (a pixel with associated depth information), we store a 2D array of layered depth pixels. A layered depth pixel stores a set of depth pixels along one line of sight sorted in front to back order. The front element in the layered depth pixel samples the first surface seen along that line of sight; the next pixel in the layered depth pixel samples the next surface seen along that line of sight, etc. When rendering from an LDI, the requested view can move away from the original LDI view and expose surfaces that were not visible in the first layer. The previously occluded regions may still be rendered from data stored in some later layer of a layered depth pixel.

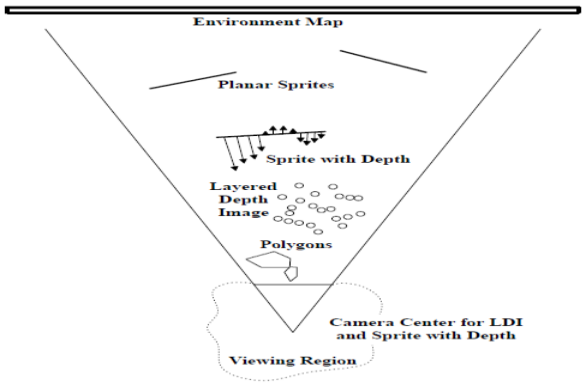


Figure 1: Different Image Based Primitives Can Serve Well Depending on Distance from the Camera

Representation grows linearly only with the depth complexity of the image. Moreover, because the LDI data are represented in a single image coordinate system, McMillan's ordering algorithm [20] can be successfully applied. As a result, pixels are drawn in the output image in back to front order allowing proper alpha blending without depth sorting. No z-buffer is required, so alpha-compositing can be done efficiently without explicit depth sorting. This makes splatting an efficient solution to the reconstruction problem.

Sprites with Depth and Layered Depth Images provide us with two new image based primitives that can be used in combination with traditional ones. Figure 1 depicts five types of primitives we may wish to use. The camera at the center of the frustum indicates where the image based primitives were generated from. The viewing volume indicates the range one wishes to allow the camera to move while still re-using these image based primitives. The assumption here is that although the part of the scene depicted in the sprite may display some parallax relative to the background environment map and other sprites, it will not need to depict any parallax within the sprite itself. Yet closer to the camera, for ele-ments with smoothly varying depth, Sprites with Depth are capable of displaying internal parallax but cannot deal with disclusions.

2. PREVIOUS WORK

A convex polygon is denoted by the tuple c = f ^ P; n; p; r; b1; b2;Bg where ^ P is the set of 3D points used to construct the convex polygon, n the number of points in P, _p the centroid of the polygon, r the normal to the polygon

plane, b1 and b2 the 2D basis vectors on the plane of the polygon and B the set of 3D points which define the convex boundary of the polygon.

Algorithm 1 Plane Filtering Algorithm

procedure PLANEFILTERING(I)
  $P, R, C, O \leftarrow \{\}$
  $n, k \leftarrow 0$
  while $n < n_{max} \wedge k < k_{max}$ **do**
    $k \leftarrow k + 1$
    $d_0 \leftarrow (\text{rand}(0, h-1), \text{rand}(0, w-1))$
    $d_1 \leftarrow d_0 + (\text{rand}(-\eta, \eta), \text{rand}(-\eta, \eta))$
    $d_2 \leftarrow d_0 + (\text{rand}(-\eta, \eta), \text{rand}(-\eta, \eta))$
    Reconstruct p_0, p_1, p_2 from d_0,d_1,d_2 ▷ eq. 1-3
    $r = \frac{(p_1 - p_0) \times (p_2 - p_0)}{\|(p_1 - p_0) \times (p_2 - p_0)\|}$ ▷ Compute plane normal
    $\bar{z} = \frac{p_{0z} + p_{1z} + p_{2z}}{3}$
    $w' = w\frac{S}{\bar{z}} \tan(f_h)$
    $h' = h\frac{S}{\bar{z}} \tan(f_v)$
    numInliers $\leftarrow 0$
    $\hat{P} \leftarrow \{\}$
    $\hat{R} \leftarrow \{\}$
    $\hat{c} \leftarrow \{\}$
    for $j \leftarrow 3, l$ **do**
      $d_j \leftarrow d_0 + (\text{rand}(-\frac{h'}{2}, \frac{h'}{2}), \text{rand}(-\frac{w'}{2}, \frac{w'}{2}))$
      Reconstruct 3D point p_j from d_j ▷ eq. 1-3
      $e = \text{abs}\,[r \cdot (p - p_0)]$
      if $e < \epsilon$ **then**
        Add p_j to $\hat{P}$
        Add r to $\hat{R}$
        numInliers $\leftarrow$ numInliers + 1
      end if
    end for
    if numInliers $> \alpha_{in} l$ **then**
      Add $\hat{P}$ to P
      Add $\hat{R}$ to R
      Construct convex polygon $\hat{c}$ from $\hat{P}$
      Add $\hat{c}$ to C
      numPoints $\leftarrow$ numPoints + numInliers
    else
      Add $\hat{P}$ to O
    end if
  end while
  return P, R, C, O
end procedure

Figure 2

3. RENDERING SPRITES

Sprites are texture maps or images with alphas (transparent pixels) rendered onto planar surfaces. They can be used either for locally caching the results of slower rendering and then generating new views by warping [30, 26, 31, 14], or they can be used directly as drawing primitives (as in video games). The texture map associated with a sprite can be computed by simply choosing a 3D viewing matrix and projecting some portion of the scene onto the image plane. In practice, a view associated with the current or expected viewpoint is a good choice. A3Dplane equation can also be computed for the sprite, e.g., by fitting a 3D plane to the z-buffer values associated with the sprite pixels. Below, we derive the equations for the 2D perspective mapping between a sprite and its novel view. This is useful both for implementing a backward mapping algorithm, and lays the foundation for our Sprites with Depth rendering algorithm. A sprite consists of an alpha-matted image $I1(x1, y1)$, a 4_4 camera matrix $C1$ which maps from 3D world coordinates (X, Y, Z, 1) into the sprite's coordinates ($x1$, $y1$, $z1$, 1),

$$\begin{bmatrix} w_1 x_1 \\ w_1 y_1 \\ w_1 z_1 \\ w_1 \end{bmatrix} = C_1 \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

($z1$ is the z-buffer value), and a plane equation. This plane equation can either be specified in world coordinates, $AX + BY + CZ + D = 0$, or it can be specified in the sprite's coordinate system, $ax1 + by1 + cz1 + d = 0$. In the former case, we can form a new camera matrix C 1 by replacing the third row of $C1$ with the row $[A\ B\ C\ D]$, while in the latter, we can compute $\hat{C}1 = PC1$, where

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ a & b & c & d \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} w_1 x_1 \\ w_1 y_1 \\ w_1 d_1 \\ w_1 \end{bmatrix} = \hat{C}_1 \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

In either case, we can write the modified projection

Where $d1 = 0$ for pixels on the plane. For pixels off the plane, $d1$ is the scaled perpendicular distance to the plane (the scale factor is 1 if $A2 + B2 + C2 = 1$) divided by the pixel to camera distance $w1$. Given such a sprite, how do we compute the 2Dtransformation associated with a novel view $\hat{C}2$? The mapping between pixels

$$\begin{bmatrix} w_2 x_2 \\ w_2 y_2 \\ w_2 \end{bmatrix} = H_{1,2} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

where $H1,2$ is the 2D planar perspective transformation (*homography*) obtained by dropping the third row and column of $T1,2$. The coordinates $(x2, y2)$ obtained after dividing out $w2$ index a pixel address in the output camera's image. Efficient backward mapping techniques exist for performing the 2D perspective warp [8, 34], or texture mapping hardware can be used.

3.1 Sprites with Depth

The descriptive power (realism) of sprites can be greatly enhanced by adding an out-of-plane displacement component d_1 at each pixel in the sprite.[1] Unfortunately, such a representation can no longer be rendered directly using a backward mapping algorithm.

Using the same notation as before, we see that the transfer equation is now

$$\begin{bmatrix} w_2 x_2 \\ w_2 y_2 \\ w_2 \end{bmatrix} = H_{1,2} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} + d_1 e_{1,2},$$

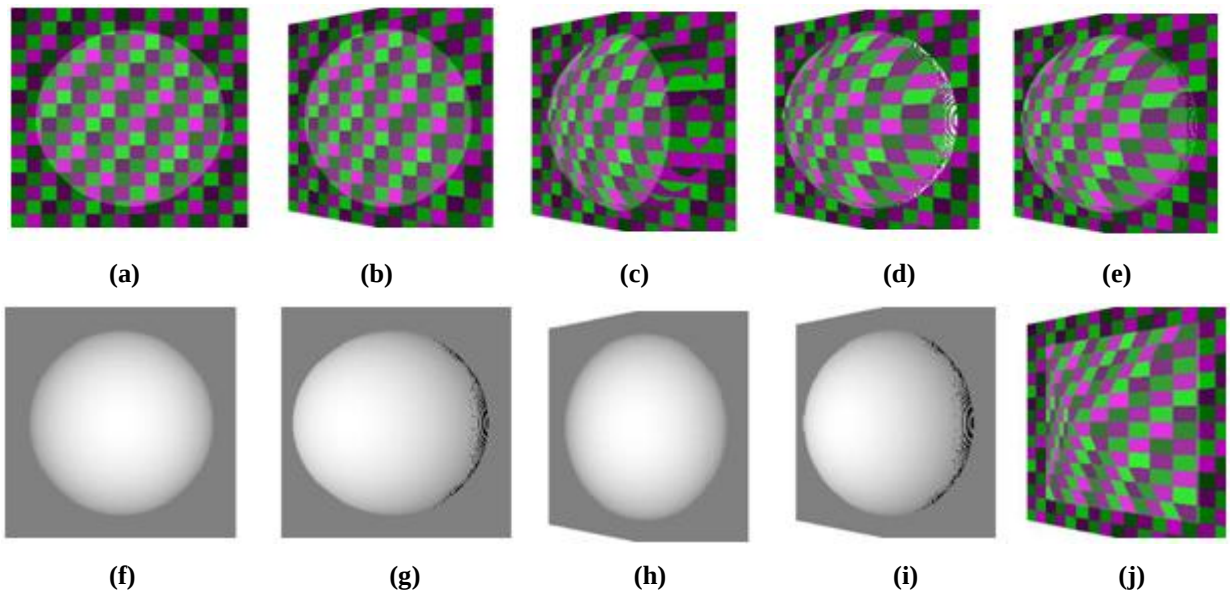
A solution to this problem is to first *forward map* the displacements *d*1, and to then use Equation (4) to perform a backward mapping step with the new (view-based) displacements. We can therefore use a quick single-pixel splat algorithm followed by a quick hole filling, or alternatively, use a simple 2 _ 2 splat. The second main advantage is that we can design the forward warping step to have a simpler form by factoring out the planar perspective warp. Notice that we can rewrite Equation (4) as

$$\begin{bmatrix} w_2 x_2 \\ w_2 y_2 \\ w_2 \end{bmatrix} = H_{1,2} \begin{bmatrix} x_3 \\ y_3 \\ 1 \end{bmatrix},$$

With

$$\begin{bmatrix} w_3 x_3 \\ w_3 y_3 \\ w_3 \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} + d_1 e^*_{1,2},$$

where $e\_1,2 = H^{-}1\ 1,2\ e1,2$.

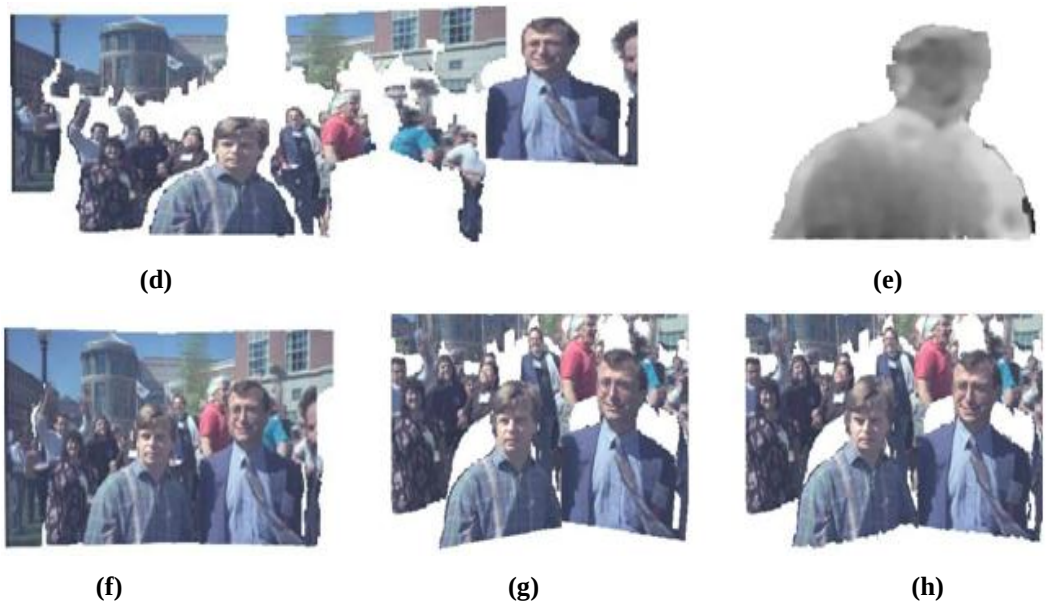


(a) (b) (c) (d) (e)

(f) (g) (h) (i) (j)

Figure 3: Plane with Bump Rendering Example: (a) Input Color (Sprite) Image I_1 (x_1 , y_1); (b) Sprite Warped by Homography Only (No Parallax); (c) Sprite Warped by Homography and Crude Parallax (d_1); (d) Sprite Warped by Homography and True Parallax (d_2); (e) With Gap Fill Width Set to 3; (f) Input Depth Map d_1 (x_1 , y_1); (g) Pure Parallax Warped Depth Map d_3 (x_3 , y_3); (h) Forward Warped Depth Map d_2 (x_2 , y_2); (i) Forward Warped Depth Map without Parallax Correction; (j) Sprite with "Pyramid" Depth Map

(a) (b) (c)



(d) **(e)**

(f) **(g)** **(h)**

Figure 4: Results of Sprite Extraction from Image Sequence: (a) Third of Five Images; (b) Initial Segmentation Into Six Layers; (c) Recovered Depth Map; (d) The Five Layer Sprites; (e) Residual Depth Image for Fifth Layer; (c) Re-Synthesized Third Image (Note Extended Field of View); (g) Novel View without Residual Depth; (h) Novel View with Residual Depth (Note the "Rounding" of the People)

Our novel two-step rendering algorithm thus proceeds in two stages:

- Forward map the displacement map $d1(x1, y1)$, using only the parallax component given in Equation to obtain $d3(x3, y3)$;
- Backward map the resulting warped displacements $d3(x3, y3)$ using Equation (5) to obtain $d2(x2, y2)$ (the displacements in the new camera view);
- Backward map the original sprite colors, using both the homography $H2,1$ and the new parallax $d2$ as in Equation (4)

(with the 1 and 2 indices interchanged), to obtain the image corresponding to camera $C2$. The last two operations can be combined into a single raster scan over the output image, avoiding the need to perspective warp $d3$ into $d2$. More precisely, for each output pixel $(x2, y2)$, we compute $(x3, y3)$ such that

$$\begin{bmatrix} w_3x_3 \\ w_3y_3 \\ w_3 \end{bmatrix} = H_{2,1} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

to compute where to look up the displacement $d3(x3, y3)$, and form the final address of the source sprite pixel using

2 We can obtain a quicker, but less accurate, algorithm by omitting the first step, i.e., the pure parallax warp from $d1$ to $d3$. If we assume the depth

3.2 Recovering Sprites from Image Sequences

While sprites and sprites with depth can be generated using computer graphics techniques, they can also be extracted from image sequences using computer vision techniques. To do this, we use a layered motion estimation

algorithm [32, 1], which simultaneously segments the sequence into coherently moving regions, and computes a parametric motion estimate (planar perspective transformation) for each layer. To convert the recovered layers into sprites, we need to determine the plane equation associated with each region.

4. LAYERED DEPTH IMAGES

While the use of sprites and Sprites with Depth provides a fast means to warp planar or smoothly varying surfaces, more general scenes require the ability to handle more general disocclusions and large amounts of parallax as the viewpoint moves. These needs have led to the development of Layered Depth Images (LDI).Like a sprite with depth, pixels contain depth values along with their colors (i.e., a *depth pixel*). In addition, a Layered Depth Image Figure 5 contains potentially multiple depth pixels per pixel location. The farther depth pixels, which are occluded from the LDI center, will act to fill in the disocclusions that occur as the viewpoint moves away from the center. The structure of an LDI is summarized by the following conceptual representation:

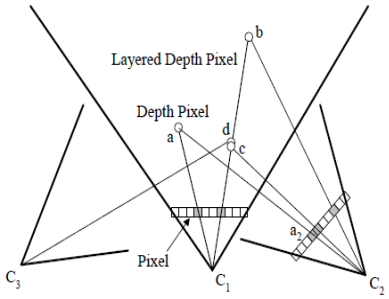


Figure 5: Layered Depth Image

In practice, we implement Layered Depth Images in two ways. When creating layered depth images, it is important to be able to efficiently insert and delete layered depth pixels, so the *Layers* array in the *Layered Depth Pixel* structure is implemented as a linked list. When rendering, it is important to maintain spatial locality of depth pixels in order to most effectively take advantage of the cache in the CPU [12]. In Section 5.1 we discuss the compact render-time version of layered depth images. There are a variety of ways to generate an LDI. Given a synthetic scene, we could use multiple images from nearby points of view for which depth information is available at each pixel. This informa-tion can be gathered from a standard ray tracer that returns depth per pixel or from a scan conversion and z-buffer algorithm where the z-buffer is also returned.

Layered Depth Image = Camera: camera

Pixels [0..xres-1,0..yres-1]: **array of** Layered Depth Pixel

The layered depth image contains camera in formation plus an array of size *x res* by *y res* layered depth pixels.

4.1 LDIs from Multiple Depth Images

We can construct an LDI by warping *n* depth images into a com-mon camera view. For example the depth images C_2 and C_3 in Figure 5 can be warped to the camera frame defined by the LDI (C_1 in figure 5). [3]If, during the warp from the input camera to the LDI camera, two or more pixels map to the same layered depth pixel, their Z values are compared. If the Z values differ by more than a preset epsilon, a new layer is added to that layered depth pixel for each distinct Z value (i.e., *Num Layers* is incremented and a new depth pixel is added), otherwise (e.g., depth pixels *c* and *d* in figure 5), the values are averaged resulting in a single depth pixel. This preprocessing is similar to the rendering described by

Max [18]. This construction of the layered depth image is effectively decou-pled from the final rendering of images from desired viewpoints. Thus, the LDI construction does not need to run at multiple frames per second to allow interactive camera motion.

4.2 LDIs from a Modified Ray Tracer

By construction, a Layered Depth Image reconstructs images of a scene well from the center of projection of the LDI (we simply display the nearest depth pixels). The quality of the reconstruction from another viewpoint will depend on how closely the distribution of depth pixels in the LDI, when warped to the new viewpoint, corresponds to the pixel density in the new image. Two common events that occur are: (1) disocclusions as the viewpoint changes, When using a ray tracer, we have the freedom to sample the scene with any distribution of rays we desire. We could simply allow the rays emanating from the center of the LDI to pierce surfaces, recording each hit along the way (up to some maximum). This would solve the disocclusion problem but would not effectively sample surfaces edge on to the LDI.

What set of rays should we trace to sample the scene, to best ap-proximate the distribution of rays from all possible viewpoints we are interested in? For simplicity, we have chosen to use a cubical region of empty space surrounding the LDI center to represent the region that the viewer is able to move in. Each face of the viewing cube defines a 90 degree frustum which we will use to define a single LDI (Figure 6). The six faces of the viewing cube thus cover all of space. For the following discussion we will refer to a single LDI. Each ray in free space has four coordinates, two for position and two for direction. Since all rays of interest intersect the cube faces, we will choose the outward intersection to parameterize the position of the ray. Direction is parameterized by two angles.

4.3 LDIs from Real Images

The dinosaur model in Figure 11 is constructed from 21 photographs of the object undergoing a 360 degree rotation on a computer-controlled calibrated turntable. An adaptation of Seitz and Dyer's voxel coloring algorithm [29] is used to obtain the LDI represen-tation directly from the input images. The regular voxelization of Seitz and Dyer is replaced by a view-centered voxelization similar to the LDI structure. The procedure entails moving outward on rays from the LDI camera center and projecting candidate voxels back into the input images. If all input images agree on a color, this voxel is filled as a depth pixel in the LDI structure. This approach en-ables straightforward construction of LDI's from images that do not contain depth per pixel.

5. RENDERING LAYERED DEPTH IMAGES

Our fast warping-based renderer takes as input an LDI along with its associated camera information. Given a new desired camera position, the warper uses an incremental warping algorithm to efficiently create an output image. Pixels from the LDI are splatted into the output image using the *over* compositing operation. The size and footprint of the splat is based on an estimated size of the re-projected pixel.

5.1 Space Efficient Representation

When rendering, it is important to maintain the spatial locality of depth pixels to exploit the second level cache in the CPU [12]. To this end, we reorganize the depth pixels into a linear array ordered from bottom to top and left to right in screen space, and back to front along a ray. We also separate out the number of layers in each layered depth pixel from the depth pixels themselves. The layered depth pixel structure does not exist explicitly in this implementation. Instead, a

double array of offsets is used to locate each depth pixel. The number of depth pixels in each scan line is accumulated into a vector of offsets to the beginning of each scan line. Within each scan line, for each pixel location, a total count of the depth pixels from the beginning of the scan line to that location is maintained. Thus to find any layered depth pixel, one simply offsets to the beginning of the scan line and then further to the first depth pixel at that location. This supports scanning in right-to-left order as well as the clipping operation discussed later.

5.2 Incremental Warping Computation

The incremental warping computation is similar to the one used for certain texture mapping operations [9, 7]. The geometry of this computation has been analyzed by McMillan [22], and efficient computation for the special case of orthographic input images is given in [3]. Let $C1$ be the 4 _ 4 matrix for the LDI camera. It is composed of an affine transformation matrix, a projection matrix, and a viewport matrix, $C1 = V1 \_P1 \_A1$. This camera matrix transforms a point from the global coordinate system into the camera's projected image coordinate system. The projected image coordinates $(x1, y1)$, obtained after multiplying the point's global coordinates by $C1$ and dividing out $w1$, index a screen pixel address. The $z1$coordinate can be used for depth comparisons in a z buffer. Let $C2$ be the output camera's matrix. Define the transfer matrix as$T1,2 = C2 _ C-1$

1. Given the projected image coordinates of some point seen in the LDI camera (e.g., the coordinates of *a* in Figure 5),

$$T_{1,2} \cdot \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} = \begin{bmatrix} x_2 \cdot w_2 \\ y_2 \cdot w_2 \\ z_2 \cdot w_2 \\ w_2 \end{bmatrix} = \mathbf{result}$$

The coordinates $(x2, y2)$ obtained after dividing by $w2$, index a pixel address in the output camera's image. Using the linearity of matrix operations, this matrix multiply can be factored to reuse much of the computation from each iteration through the layers of a layered depth pixel; **result** can be computed

As

$$T_{1,2} \cdot \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} = T_{1,2} \cdot \begin{bmatrix} x_1 \\ y_1 \\ 0 \\ 1 \end{bmatrix} + z_1 \cdot T_{1,2} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \mathbf{start} + z_1 \cdot \mathbf{depth}$$

To compute the warped position of the next layered depth pixel along a scanline, the new **start** is simply incremented

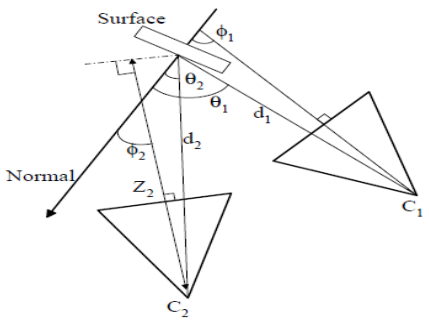


Figure 6: Values for Size Computation of a Projected Pixel

$$T_{1,2} \cdot \begin{bmatrix} x_1 + 1 \\ y_1 \\ 0 \\ 1 \end{bmatrix} = T_{1,2} \cdot \begin{bmatrix} x_1 \\ y_1 \\ 0 \\ 1 \end{bmatrix} + T_{1,2} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{start} + \mathbf{xincr}$$

The warping algorithm proceeds using McMillan's ordering algorithm [20]. The LDI is broken up into four regions above and below and to the left and right of the epipolar point. For each quadrant, the LDI is traversed in (possibly reverse) scan line order. At the beginning of each scan line, **start** is computed. The sign of **xincr**is determined by the direction of processing in this quadrant. Each layered depth pixel in the scan line is then warped to the output image by calling *Warp*. This procedure visits each of the layers in back to front order and computes **result** to determine its location in the output image. As in perspective texture mapping, a divide is required per pixel. Finally, the depth pixel's color is splatted at this location in the output image

The following pseudo code summarizes the warping algorithm applied to each layered depth pixel.

```
procedure Warp(ldpix, start, depth, xincr)
    for k ←0 to dpix.NumLayers-1
        z1 ←ldpix.Layers[k].Z
        result ←start + z1 * depth
        //cull if the depth pixel goes behind the output camera
        //or if the depth pixel goes out of the output cam's frustum
        if result.w > 0 and IsInViewport(result) then
            result ←result / result.w
            // see next section
            sqrtSize ←z2 * lookupTable[ldpix.Layers[k].SplatIndex]
            splat(ldpix.Layers[k].ColorRGBA, x2, y2, sqrtSize)
        end if
        // increment for next layered pixel on this scan line
        start ←start + xincr
    end for
end procedure
```

5.3 Splat Size Computation

To splat the LDI into the output image, we estimate the projected area of the warped pixel. This is a rough approximation to the foot print evaluation [33] optimized for speed. The proper size can be computed (differentially) as

$$size = \frac{(d_1)^2 \cos(\theta_2)\, res_2 \tan(fov_1/2)}{(d_2)^2 \cos(\theta_1)\, res_1 \tan(fov_2/2)}$$

where $d1$ is the distance from the sampled surface point to the LDI camera, $fov1$ is the field of view of the LDI camera, $res1 = (w1h1)-1$ where $w1$ and $h1$ are the width and height of the LDI, and _1 is the angle between the surface normal and the line of sight to the LDI camera (see Figure 6). The same terms with subscript 2 refer to the output camera.

It will be more efficient to compute an approximation of the square root of size,

$$\sqrt{size} = \frac{1}{d_2} \cdot \frac{d_1 \sqrt{cos(\theta_2) res_2 tan(fov_1/2)}}{\sqrt{cos(\theta_1) res_1 tan(fov_2/2)}}$$
$$\approx \frac{1}{Z_2} \cdot \frac{d_1 \sqrt{cos(\phi_2) res_2 tan(fov_1/2)}}{\sqrt{cos(\phi_1) res_1 tan(fov_2/2)}}$$
$$\approx z_2 \cdot \frac{d_1 \sqrt{cos(\phi_2) res_2 tan(fov_1/2)}}{\sqrt{cos(\phi_1) res_1 tan(fov_2/2)}}$$

We approximate the _s as the angles _ between the surface normal vector and the *z* axes of the camera's coordinate systems. We also approximate *d*2 by *Z*2, the *z* coordinate of the sampled point in the output camera's unprojected eye coordinate system. During rendering, we set the projection matrix such that *z*2 = 1=*Z*2. The current implementation supports 4 different splat sizes, so a very crude approximation of the size computation is implemented using a lookup table. For each pixel in the LDI, we store *d*1 using 5 bits. We use 6 bits to encode the normal, 3 for *nx*, and 3 for *ny*. This gives us an eleven-bit lookup table index. Before rendering each new image, we use the new output camera information to precompute values for the 2048 possible lookup table indexes. At each pixel we obtain *psize* by multiplying the computed *z*2 by the value found in the lookup table.

$$\sqrt{size} \approx z_2 \cdot \texttt{lookup[nx, ny, d1]}$$

To maintain the accuracy of the approximation for *d*1, we discretize *d*1 nonlinearly using a simple exponential function that allocates more bits to the nearby *d*1 values, and fewer bits to the distant *d*1 values. The four splat sizes we currently use have 1 by 1, 3 by 3, 5 by 5, and 7 by 7 pixel footprints. Each pixel in a footprint has an alpha value to approximate a Gaussian splat kernel. However, the alpha values are rounded to 1, 1/2, or 1/4, so the alpha blending can be done with integer shifts and adds

5.4 Depth Pixel Representation

The size of a cache line on current Intel processors (Pentium Pro and Pentium II) is 32 bytes. To fit four depth pixels into a single cache line we convert the floating point Z value to a 20 bit integer. This is then packed into a single word along with the 11 bit splat table index. These 32 bits along with the R, G, B, and alpha values fill out the 8 bytes. This seemingly small optimization yielded a 25 percent improvement in rendering speed.

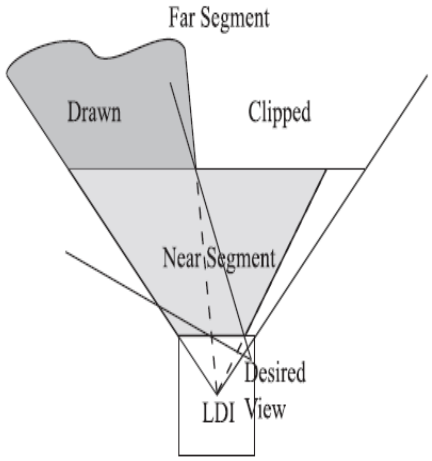


Figure 7: LDI with Two Sements

5.5 Clipping

The LDI of the chestnut tree scene in Figure 11 is a large data set containing over 1.1 million depth pixels. If we naively render this LDI by reprojecting every depth pixel, we would only be able to render at one or two frames per second. When the viewer is close to the tree, there is no need to flow those pixels that will fall outside of the new view. Unseen pixels can be culled by intersecting the view frustum with the frustum of the LDI. This is implemented by intersecting the view frustum with the near and far plane of the LDI frustum, and taking the bounding box of the intersection. This region defines the rays of depth pixels that could be seen in the new view. This computation is conservative, and gives suboptimal results when the viewer is looking at the LDI from the side (see Figure 7). The view frustum intersects almost the entire cross section of the LDI frustum, but only those depth pixels in the desired view need be warped. Our simple clipping test indicates that most of the LDI needs to be warped. To alleviate this, we split the LDI into two segments, a near and a far segment (see Figure 7). These are simply two frustra stacked one on top of the other. The near frustum is kept smaller than the back segment. We clip each segment individually, and render the back segment first and the front segment second. Clipping can speed rendering times by a factor of 2 to 4.

6. RESULTS

While the LDIs are allocated with a maximum of 10 layers per pixel, the average depth complexity for these LDIs is only 1.24. Thus the use of three input images only increases the rendering cost by 24 percent. renderer (running concurrently in a high-priority thread) generates images at 300 by 300 resolution. On a Pentium II PC running at 300MHz, we achieved frame rate of 6 to 10 frames per second. Figures 9 and 10 show two cross-eye stereo pairs of a chestnut tree. In Figure 9 only the near segment is displayed. Figure 9 shows both segments in front of an environment map. The LDIs were created using a modified version of the Rayshade raytracer. The tree model is very large; Rayshade allocates over 340 MB of memory to render a single image of the tree. The stochastic method discussed in Section 4.2 took 7 hours to trace 16 million rays through this scene using an SGI Indigo2 with a 250 MHz processor and 320MB of memory. The resulting LDI has over 1.1 million depth pixels, 70,000 of which were placed in the near segment with the rest in the far segment. When rendering this interactively we attain frame rates between 4 and 10 frames per second on a Pentium II PC running at 300MHz.

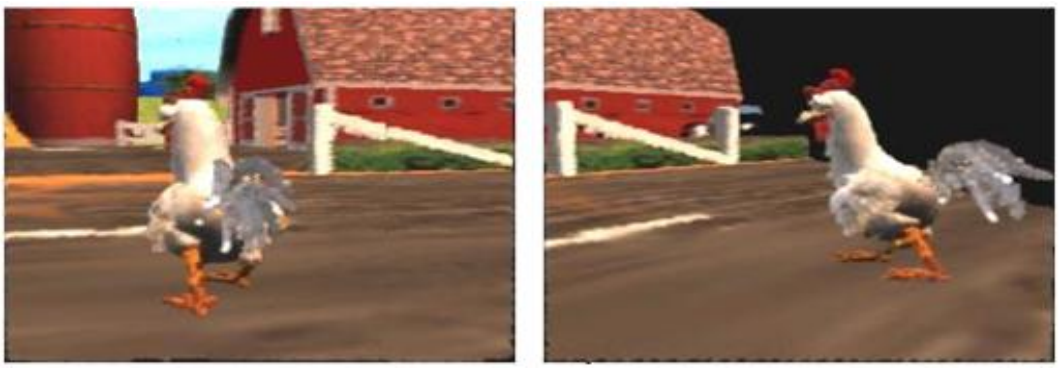


Figure 8: Barnyard Scene

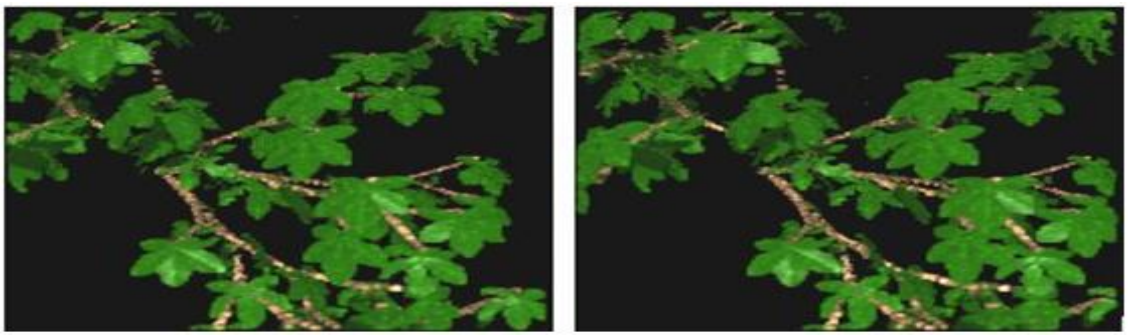


Figure 9: Near Segment of Chestnut

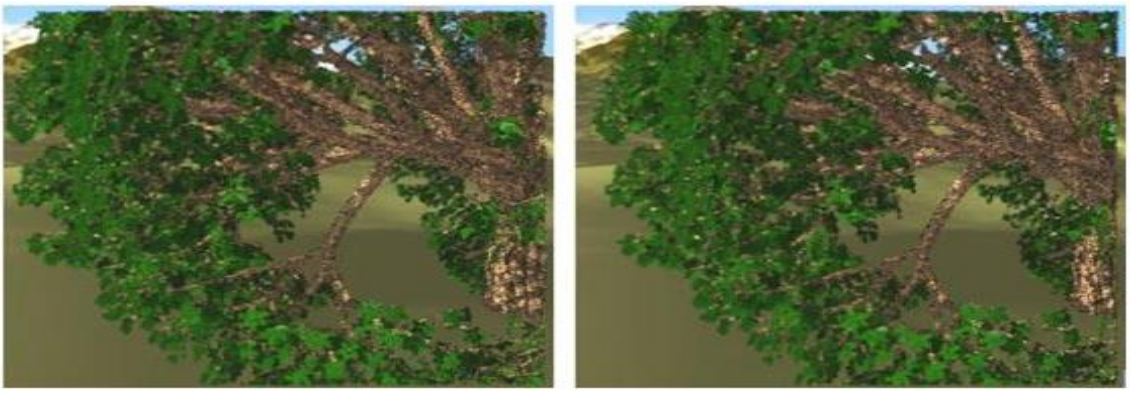


Figure 10: Chestnut Tree in Front of Environment

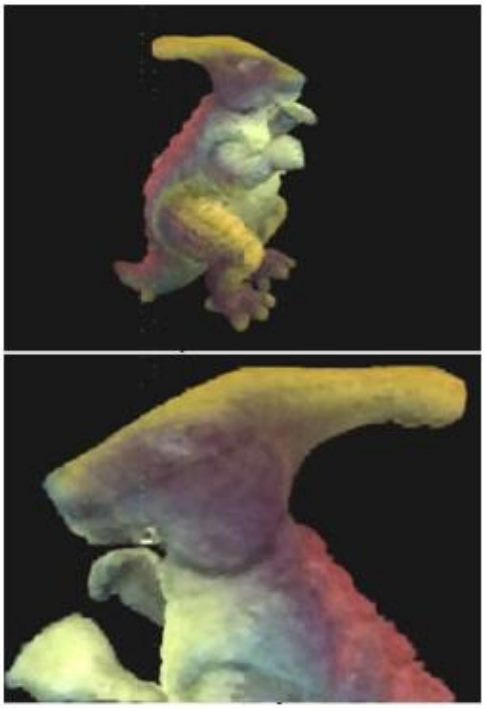


Figure 11: Dinosaur Model Reconstructed from 21 Photographs

7. DISCUSSIONS

In this paper, we have described two novel techniques for image based rendering. The first technique renders Sprites with Depth without visible gaps, and with a smoother rendering than traditional forward mapping (splatting)

techniques. It is based on the observa-tion that a forward mapped displacement map does not have to be as accurate as a forward mapped color image. If the displacement map is smooth, the inaccuracies in the warped displacement map result in only sub-pixel errors in the final color pixel sample positions.

Our second novel approach to image based rendering is a Layered Depth Image representation. The LDI representation provides the means to display the parallax induced by camera motion as well as reveal disoccluded regions. The average depth complexity in our LDI's is much lower that one would achieve using multiple input images (e.g., only 1.24 in the Chicken LDI). The LDI representation takes advantage of McMillan's ordering algorithm allowing pixels to be splatted back to Front with an *over* compositing operation. Traditional graphics elements and planar sprites can be combined with Sprites with Depth and LDIs in the same scene if a back-to-front ordering is maintained. In this case they are simply composited onto one another. Without such an ordering a z-buffer approach will still work at the extra cost of maintaining depth information per frame.

Choosing a single camera view to organize the data has the advan-tage of having sampled the geometry with a preference for views very near the center of the LDI. This also has its disadvantages. First, pixels undergo two resampling steps in their journey from in-put image to output. This can potentially degrade image quality. Secondly, if some surface is seen at a glancing angle in the LDIs view the depth complexity for that LDI increases, while the spatial sampling resolution over that surface degrades. The sampling and aliasing issues involved in our layered depth image approach are still not fully understood; a formal analysis of these issues would be helpful.

With the introduction of our two new representations and rendering techniques, there now exists a wide range of different image based rendering methods available. At one end of the spectrum are tradi-tional texture-mapped models. When the scene does not have too much geometric detail, and when texture-mapping hardware is avail-able, this may be the method of choice. If the scene can easily be partitioned into non-overlapping sprites (with depth), then triangle-based texture-mapped rendering can be used without requiring a z buffer [17, 4].

All of these representations, however, do not explicitly account for certain variation of scene appearance with viewpoint, e.g., specu-larities, transparency, etc. View-dependent texture maps [5], and 4D representations such as lightfields or Lumigraphs [15, 7], have been designed to model such effects. These techniques can lead to greater realism than static texture maps, sprites, or Layered Depth Images, but usually require more effort (and time) to render.

In future work, we hope to explore representations and rendering al-gorithms which combine several image based rendering techniques. Automatic techniques for taking a 3D scene (either synthesized or real) and re-representing it in the most appropriate fashion for im-age based rendering would be very useful. These would allow us to apply image based rendering to truly complex, visually rich scenes, and thereby extend their range of applicability.

REFERENCES

1. S. Baker, R. Szeliski, and P. Anandan. A Layered Approach to Stereo Reconstruction. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'98)*. Santa Barbara, June 1998.
2. Shenchang Eric Chen and Lance Williams. View Interpolation for Im-age Synthesis. In James T. Kajiya, editor, *Computer Graphics (SIG-GRAPH '93 Proceedings)*, volume 27, pages 279–288. August 1993.

3. William Dally, Leonard McMillan, Gary Bishop, and Henry Fuchs. The Delta Tree: An Object Centered Approach to Image Based Rendering. AI technical Memo 1604, MIT, 1996.
4. Lucia Darsa, Bruno Costa Silva, and Amitabh Varshney. Navigating Static Environments Using Image-Space Simplification and Morphing. In *Proc. 1997 Symposium on Interactive 3D Graphics*, pages 25–34. 1997.
5. Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and Rendering Architecture from Photographs: A Hybrid Geometry-and Image-Based Approach. In Holly Rushmeier, editor, *SIGGRAPH* 96 *Conference Proceedings*, Annual Conference Series, pages 11–20. ACM SIGGRAPH, Addison Wesley, August 1996.
6. O. Faugeras. *Three-dimensional computer vision: A geometric view-point*. MIT Press, Cambridge, Massachusetts, 1993.
7. Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The Lumigraph. In Holly Rushmeier, editor, *SIGGRAPH* 96 *Conference Proceedings*, Annual Conference Series, pages 43–54. ACM SIGGRAPH, Addison Wesley, August 1996.
8. Paul S. Heckbert. Survey of Texture Mapping. *IEEE Computer Graph-ics and Applications*, 6(11): 56–67, November 1986.
9. Paul S. Heckbert and Henry P. Moreton. Interpolation for Polygon Texture Mapping and Shading. In David Rogers and Rae Earnshaw, editors, *State of the Art in Computer Graphics: Visualization and Mod-eling*, pages 101–111. Springer-Verlag, 1991.
10. Youichi Horry, Ken ichi Anjyo, and Kiyoshi Arai. Tour Into the Pic-ture: Using a Spidery Mesh Interface to Make Animation from a Single Image. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceed-ings*, Annual Conference Series, pages 225–232. ACM SIGGRAPH, Addison Wesley, August 1997.
11. R. Kumar, P. Anandan, and K. Hanna. Direct recovery of shape from multiple views: A parallax based approach. In *Twelfth International Conference on Pattern Recognition (ICPR'94)*, volume A, pages 685– 688 IEEE Computer Society Press, Jerusalem, Israel, October 1994.
12. Anthony G. LaMarca. Caches and Algorithms. Ph.D. thesis, University of Washington, 1996.
13. S. Laveau and O. D. Faugeras. 3-D Scene Representation as a Col-lection of Images. In *Twelfth International Conference on Pattern Recognition (ICPR'94)*, volume A, pages 689–691. IEEE Computer Society Press, Jerusalem, Israel, October 1994.
14. Jed Lengyel and John Snyder. Rendering with Coherent Layers. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, An-nual Conference Series, pages 233–242. ACM SIGGRAPH, Addison Wesley, August 1997.
15. Marc Levoy and Pat Hanrahan. Light Field Rendering. In Holly Rush-meier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Confer-ence Series, pages 31–42. ACM SIGGRAPH, Addison Wesley, August 1996.
16. Mark Levoy and Turner Whitted. The Use of Points as a Display Primitive. Technical Report 85-022, University of North Carolina, 1985.

17. William R. Mark, Leonard McMilland, and Gary Bishop. Post-Rendering 3D Warping. In *Proc. 1997 Symposium on Interactive 3D Graphics*, pages 7–16. 1997.

18. Nelson Max. Hierarchical Rendering of Trees from Precomputed Multi-Layer Z-Buffers. In Xavier Pueyo and Peter Schroder,¨ editors, *Eurographics Rendering Workshop 1996*, pages 165–174. Eurograph-ics, Springer Wein, New York City, NY, June 1996.

19. Leonard McMillan. Computing Visibility Without Depth. Technical Report 95-047, University of North Carolina, 1995.

20. Leonard McMillan. A List-Priority Rendering Algorithm for Redis-playing Projected Surfaces. Technical Report 95-005, University of North Carolina, 1995.

21. Leonard McMillan and Gary Bishop. Plenoptic Modeling: An Image-Based Rendering System. In Robert Cook, editor, *SIGGRAPH 95 Con-ference Proceedings*, Annual Conference Series, pages 39–46. ACM SIGGRAPH, Addison Wesley, August 1995.

22. Leonard McMillan and Gary Bishop. Shape as a Pertebation to Projec-tive Mapping. Technical Report 95-046, University of North Carolina, 1995.

23. Don P. Mitchell. *personal communication*. 1997.

24. Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Render-ing Complex Scenes with Memory-Coherent Ray Tracing. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Con-ference Series, pages 101–108. ACM SIGGRAPH, Addison Wesley, August 1997.

25. H. S. Sawhney. 3D Geometry from Planar Parallax. In *IEEE Com-puter Society Conference on Computer Vision and Pattern Recognition (CVPR'94)*, pages 929–934. IEEE Computer Society, Seattle, Wash-ington, June 1994.

26. Gernot Schaufler and Wolfgang St urzlinger¨. A Three-Dimensional Image Cache for Virtual Reality. In *Proceedings of Eurographics '96*, pages 227–236. August 1996.

27. Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul E. Haeberli. Fast shadows and lighting effects using texture mapping. In Edwin E. Catmull, editor, *Computer Graphics (SIGGRAPH '92 Proceedings)*, volume 26, pages 249–252. July 1992.

28. Steven M. Seitz and Charles R. Dyer. View Morphing: Synthesizing 3D Metamorphoses Using Image Transforms. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 21–30. ACM SIGGRAPH, Addison Wesley, August 1996.

29. Steven M. seitz and Charles R. Dyer. Photorealistic Scene Recon-struction by Voxel Coloring. In *Proc. Computer Vision and Pattern Recognition Conf.*, pages 1067–1073. 1997.

30. Jonathan Shade, Dani Lischinski, David Salesin, Tony DeRose, and John Snyder. Hierarchical Image Caching for Accelerated Walk-throughs of Complex Environments. In Holly Rushmeier, editor, *SIG-GRAPH 96 Conference Proceedings*, Annual Conference Series, pages 75–82. ACM SIGGRAPH, Addison Wesley, August 1996.